Functional Programming Individual Project Report: Threads and Concurrent Computation

Introduction

This project simulates client-server interactions using Haskell, focusing on concurrency and thread-safe communication. It integrates clients, a server, a REST API, and an SQLite database for managing requests and responses. Clients generate requests that are processed by the server, ensuring all interactions are logged for traceability. The REST API provides real-time insights into the system, while `MVar` ensures safe concurrency management.

Issues Faced

Managing concurrency among multiple threads accessing shared resources was a primary challenge. Race conditions and potential deadlocks were mitigated using `modifyMVar_` for atomic operations. SQLite's "database is locked" errors due to concurrent writes were resolved with a retry mechanism (`withRetry`). Ensuring consistency between requests and their corresponding responses was addressed by assigning unique `requestId`s. Additionally, implementing real-time updates via the REST API required careful synchronization of shared resources like the request queue, handled using `MVar`.

Design Decisions

The Request and Response types were designed to encapsulate essential data, including `requestId`, `requestTime`, and `requestContent`, ensuring traceability. These types derive from `ToJSON` and `FromJSON` for seamless serialization in the REST API and log files. The system extensively uses `MVar` for concurrency control. The request queue, modeled as `RequestQueue = MVar [Request]`, ensures safe enqueuing and dequeuing of requests. A global counter (`MVar Int`) tracks unique `requestId`s across clients, and an additional `MVar ()` serializes database access to prevent race conditions during writes.

The Main module initializes the system, spawns client threads, and ensures all requests are processed before saving logs. The Client module generates unique requests and enqueues them while persisting them in the database. The Server module processes requests, generates responses, and ensures consistency by saving them in the database using synchronized operations. The Database module handles persistent storage, retrying operations when concurrency issues occur, ensuring data integrity. Finally, the RestAPI module, implemented with Scotty, provides endpoints to manage requests dynamically, view the queue state, and retrieve logs, ensuring real-time interaction.

Conclusion

This project effectively demonstrates Haskell's capabilities in concurrency management, real-time RESTful interaction, and storage using SQLite. Challenges like database concurrency issues, thread safety, and data consistency were addressed through thoughtful design and the careful use of `MVar`. The system achieves reliability, scalability, and usability, making it a solution for concurrent computations.

Zeynep Kurtulus
ID: 240037253